# DOSSIER DE PROJET

Marie Germain

Développeur Web et Web mobile

Studi

Application mobile Skimtrack

Skimtrack

# Remerciements

Je tiens à remercier Madame Béland Laure qui m’a fait confiance et a cru en moi pour mener à bien son projet. Elle m'a donné l'opportunité de participer à un projet intéressant par sa complexité, qui m'a permis de me confronter à la réalité du métier de développeur, de mettre en pratique les connaissances acquises lors de la formation et d’en acquérir beaucoup d’autres durant ce stage.
Je la remercie également pour son accueil. Elle m’a intégrée à l’équipe, comme faisant partie de la « Team Skimtrack », au sein de laquelle l’avis de chacun est pris en compte.

Je souhaite remercier Emma, stagiaire UX en formation chez le Village Du Code à Bordeaux, qui m'a fait découvrir l'UX design, et avec qui j'ai pu partager durant nos deux mois de stage en commun.

Je tiens à remercier mes collègues de formation avec qui j'ai pu échanger autour de mon projet et qui m'ont apporté leurs conseils quand j'en avais besoin, ainsi que mon formateur pour sa disponibilité, son soutien, ainsi que pour les conseils techniques qu'il m'apporte tout au long de mon stage.

# Sommaire


Remerciements 2

Sommaire 3

Introduction 4

1. Compétences du référentiel couvertes par le projet 4

a. Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité 4

b. Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité 4

2. Résumé du projet 5

3. Environnement humain et technique 6

a. Environnement humain 6

b. Environnement technique 6

Le projet de stage 7

1. Le cahier des charges 7

2. Spécifications techniques 8

La réalisation du projet 9

1. Conception de la base de données 9

2. Développement de l'API 11

a. Installation 11

b. Initialisation du projet 12

c. Développement de la base de données 13

d. Développement des routers et controllers 14

3. Développement de la partie front-end 16

a. Mise en place de l'environnement de développement 16

b. La navigation au sein de l'application 16

c. Les fonts 18

d. Les contextes 19

e. Écrans développés 20

Fonctionnalité la plus représentative: L'authentification 24

1. Jeu d'essai 24

2. Recherches anglophones sur le JWT 28

Conclusion 29

Annexes 30

# Introduction

## 1. Compétences du référentiel couvertes par le projet

### a. Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité

- Réaliser une interface utilisateur web statique et adaptable
- Développer une interface utilisateur web dynamique

L'application mobile doit être disponible sur Android et iOS, d'où le choix de développer en React Native qui permet un développement cross-plateform. Malgré cela, il est nécessaire d'effectuer des ajustements suivant le système d'exploitation. Elle doit pouvoir proposer une expérience utilisateur similaire quelle que soit la taille de l'écran. J'ai donc fait en sorte que l'interface s'adapte aux différentes tailles de mobiles.

### b. Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité

- Créer une base de données
- Développer les composants d'accès aux données
- Développer la partie back-end d'une application web ou web mobile

La partie front fait appel à une API qui renvoie les données d'une base de données relationnelle que j'ai pensée et développée. Concernant l'API, je l'ai développée en Node.js, grâce au framework Express.js. De plus, elle utilise Knex.js, un query builder qui permet de développer facilement une base de données en JavaScript, et Objection.js, un ORM (Object-Relational Mapping) pour Node.js.

## 2. Résumé du projet

Dans le cadre de ma formation Développeur Web – Web Mobile, je réalise un stage de 4 mois qui a débuté le 17 août 2020, et ce jusqu'au 17 décembre 2020, dans le cadre du projet « Skimtrack ».

Ma maître de stage, Madame Béland Laure porte ce projet depuis mars 2020 au sein d'un incubateur.

Le projet Skimtrack est une application mobile de parcours sportif. L'utilisateur choisit un parcours en plein air et accède à du contenu vidéo disponible à chaque point d'intérêt sur une carte. Ces vidéos courtes montrent des coachs présentant les exercices à pratiquer sur le lieu. Le contenu Skimtrack est accessible via l'achat de forfait mensuel directement dans l'application.

Le déploiement d'une première version de l'application avec les premières fonctionnalités est prévu au printemps 2021. Mon rôle au sein du projet est de procéder à l'ensemble du développement de l'application mobile : React Native pour le front, Node.js et Express pour le développement de l'API, ainsi que la mise en place de la base de données relationnelle.

Étant la seule développeuse, je réalise ce stage en totale autonomie au sein de mon équipe. Cela m'a permis de mettre en pratique les compétences acquises lors de la formation, mais également d'en gagner de nouvelles très rapidement en me référant aux différentes documentations et forums dédiés, que j'ai pu appliquer au projet, qui était au début du stage, un véritable challenge pour moi.

# 3. Environnement humain et technique

## a. Environnement humain

Lorsque j'ai rejoint l'équipe Skimtrack, elle n'était jusqu'alors composée que de Madame Béland.

Madame Béland est l'initiatrice du projet Skimtrack. Elle s'occupe plutôt de la partie technique, de trouver le meilleur moyen d'aborder les différentes fonctionnalités de l'application, de trouver des partenaires dans le milieu du sport afin de donner de la visibilité au projet. Bien qu'elle ne soit pas formée sur les technologies que j'utilise, son expérience dans le développement informatique m'incite à prendre du recul lorsque je rencontre un obstacle et m'aide parfois dans le raisonnement qui m'amène à trouver la solution.

Début septembre, nous avons été rejointes par Emma, stagiaire UX designer, issue de la formation UX dispensée au sein du Village du Code à Bordeaux. Le travail d'Emma a eu un impact fort sur mon stage. Grâce à elle, j'ai pris conscience de l'importance de l'UX au sein d'un projet, j'ai appris les différentes étapes d'un parcours UX, en participant notamment aux différentes étapes de la réflexion autour d'une fonctionnalité ciblée.

Puis, début novembre, nous avons été rejointes par Thomas, étudiant en licence professionnelle Création et Diffusion Audiovisuelle sur Internet. Il s'occupe entièrement de la création des vidéos qui auront une identité « Skimtrack ».

## b. Environnement technique

Je travaille sur un ordinateur portable car j'ai besoin d'être mobile. En effet, je me rends à l'incubateur 2 jours par semaine où je travaille avec les membres de l'équipe. Les 3 jours restants je travaille au sein de l'espace de coworking de la Station à Bordeaux avec d'autres apprentis développeurs, qui sont également en télétravail. C'est l'occasion pour moi d'échanger avec eux, d'évoquer les obstacles auxquels je suis confrontée et parfois trouver la solution à travers ces échanges.

# Le projet de stage

## 1. Le cahier des charges

Une première application web Skimtrack a été développée par des étudiants du Village du Code de Bordeaux dans le cadre de leur formation. Elle est développée en ReactJS pour le front qui fait appel à une API développée en Node.js qui interroge une base de données relationnelle. Cette application ne comprenait que très peu de fonctionnalités. Elle m'a surtout servi de prototype initial, notamment pour la partie front. J'ai effectué le déploiement de cette application à mon arrivée en stage. La partie back et la base de données sont déployées sur mon serveur privé, et la partie front est déployée sur la plateforme Netlify.

Le cahier des charges de l'application mobile à développer était déjà très complet à mon arrivée en stage. Les fonctionnalités se sont allégées au fur et à mesure de l’avancement des travaux UX (maquettes, interview, mises à jour).
Nous avons donc travaillé en mode agile avec une méthodologie Scrum allégée. J’ai ainsi adapté l’avancement du développement en fonction des fonctionnalités et écrans validés au fur et à mesure.

A son arrivée dans l'application, l'utilisateur arrive sur un écran d'onBoarding, où est expliqué le principe de l'application. Il peut ensuite choisir de s'inscrire ou de s'authentifier.

L'utilisateur accède à un mode "parcours libre", il est géolocalisé sur une carte, où apparaissent des marqueurs, indiquant les points d'intérêt se trouvant à proximité.
L'utilisateur peut choisir d'accéder à l'onglet "parcours guidés", où il trouvera la liste des parcours à proximité, présentés sous forme d'une liste scrollable de vignettes, qui contiennent diverses informations à propos de chaque parcours.
Si l'utilisateur est abonné, il peut débuter un parcours, pendant lequel il sera guidé de point d'intérêt en point d'intérêt, grâce à un système de GPS. Chaque point d'intérêt possède une ou plusieurs vidéos d’exercices de renforcement musculaire.
A la fin de chaque parcours, l'utilisateur peut laisser un commentaire et peut partager son expérience sur les réseaux sociaux.
L'utilisateur a également accès son profil utilisateur, où il peut ajouter un avatar, modifier ses informations personnelles, visualiser ses parcours marqués comme favoris ou se déconnecter.

L'abonnement se fait “In app” grâce au compte Google Play ou Apple.

L'utilisateur peut activer d'autres fonctionnalités :

- Activer les notifications afin d'être informé lorsqu'il se trouve près d'un point d'intérêt
- Activer ou non le tracking d'activité cardiaque grâce à un bracelet connecté vendu séparément.

## 2. Spécifications techniques

Voici les différentes technologies que j'ai utilisé pour ce projet :

- Environnement de travail :
  - Le framework-plateforme **Expo** apporte des outils et des services qui facilitent le développement d'applications React Native, notamment avec la possibilité d'avoir un rendu instantané de notre application sur un véritable mobile grâce à l'application éponyme.
  - L'outil de versioning **Git**, qui me permet de développer les fonctionnalités sur une nouvelle branche, en plus d'utiliser **Github** pour y déposer mon code après chaque commit.
  - Le logiciel **Postman** qui permet d'effectuer des tests auprès de l'API pendant son développement.

- Partie Front :
  - **React Native** : framework gratuit et Open Source créé par Facebook, basé sur la librairie JavaScript React. Il permet le développement d'applications cross-plateforms et utilise les composants mobiles natifs.

- Partie Back :
  - **Node.js** : un environnement bas niveau permettant l'exécution de code JavaScript côté serveur. Grâce à son fonctionnement non bloquant, il permet de concevoir des applications en réseau performantes, telles qu'un serveur web ou une API.
  - **Express.js** : framework qui permet de construire des applications web basées sur Node.js, pratique pour le développement de serveur. Il permet une création d'API robuste de manière simple et rapide.
  - **Knex.js** : SQL builder qui permet de créer une base de données, de la modifier et d'écrire des requêtes avec une syntaxe objet.
  - **Objection.js** : ORM (Object Relational Mapping) qui fonctionne avec Knex.js. Son système de modèles permet une description précise des données attendues.

# La réalisation du projet

## 1. Conception de la base de données

Avec Madame Béland, nous avons commencé par lister toutes les données que nous avions besoin de stocker. Nous nous sommes posées la question suivante : quelles entités avons-nous au sein de l'application ? Par exemple, nous avons des parcours. Un parcours est caractérisé par un id, un nom, une description, une ou plusieurs vidéos d'exercice appelées capsules, un potentiel coût supplémentaire si dans un terrain privé et l'id de l'utilisateur qui a ajouté le parcours. Une fois toutes nos entités définies, nous avons utilisé la méthode Merise, méthode d'analyse et de conception de systèmes d'informations, notamment de bases de données.

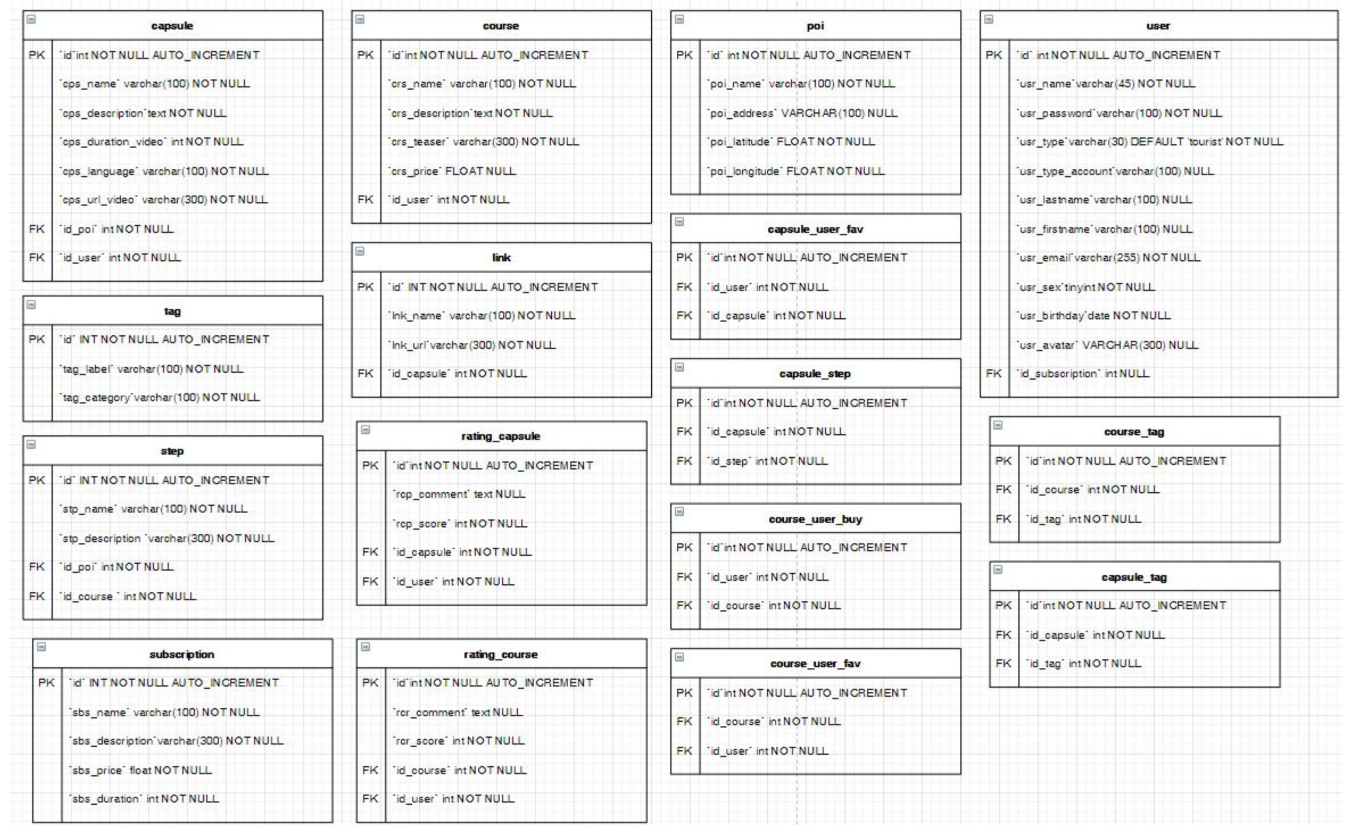
Nous avons commencé par établir le Modèle Conceptuel de Données (MCD). Cette étape consiste à établir des relations entre les différentes entités, en précisant les cardinalités.

Modèle conceptuel de données

Une fois ces dernières établies, nous avons pu passer à l'élaboration du Modèle Logique de Données (MLD). Chaque entité devient une table et leur identifiant devient la clé primaire de la table. Pour les relations ayant des cardinalités *,1 — *,n la clé primaire de la table côté *,n devient une clé étrangère de la table côté *,1 créant ainsi la relation entre les tables. Les relations "Many to Many", représentées par des cardinalités *,n — *,n , génèrent une nouvelle table qui comportera 2 champs, correspondant aux id de chacune des 2 entités.

Le modèle logique de données

Ensuite, nous avons pu élaborer le Modèle Physique de Données (MPD), qui consiste à préciser le type et les possibles restrictions appliquées pour chaque champ qui compose les tables.
Une fois l'élaboration de la structure de la base de données terminée, j'ai procédé au développement de la partie back, afin de générer dans un premier temps, la base de données.



Le modèle physique de données

## 2. Développement de l’API

### a. Installation

J'ai donc créé un dossier dans lequel j'ai lancé la commande *npm init* qui initialise un projet en créant le fichier package.json, qui contient les informations du projet : le nom, la version, la description, le fichier qui sera le point d'entrée, les commandes qui serviront aux tests, le repository git associé, des mots-clés, l'auteur et la licence.

Node.js étant déjà installé sur mon ordinateur, j'ai pu passer à l'installation d'Express, grâce à la commande : *npm install express --save.*

Puis à l'installation de Knex et Objection:
*npm install knex objection --save*

Puis de la librairie correspondant à la base de données :
*npm install mysql2 –save*

## b. Initialisation du projet

Une fois l'installation de toutes les librairies effectuée, j'ai créé la base de données locale "Skimtrack" en utilisant phpMyAdmin. J'ai également créé un nouveau compte utilisateur, en lui attribuant tous les privilèges sur cette base de données.

Dans le dossier du projet, j'ai créé un fichier .env qui contient notamment les variables nécessaires à la connexion à la base de données : le host, l'utilisateur, le mot de passe, le nom de la base de données, ainsi que le port utilisé. L'installation de la libraire « dotenv » grâce à la commande : *npm install dotenv* puis son import dans le fichier server.js (qui sera le point d'entrée) nous permet d'accéder aux variables situées dans le fichier .env depuis n'importe où, grâce à : process.env.[nom_de_la_variable].

Avec la commande *npx knex init*, je peux ensuite générer le fichier knexfile.js, qui contiendra les paramètres nécessaires à la connexion à la base de données pendant le développement.

```
module.exports = {

  development: {
    client: 'mysql2',
    connection: () => ({
      host : process.env.DB_HOST,
        user : process.env.DB_USER,
        password : process.env.DB_PASSWORD,
        database : process.env.DB_DATABASE

    }),
    debug: true
  }
};
```

Fichier knexfile.js

Ensuite, il me faut importer Express, Knex et Objection dans le fichier server.js puis d'indiquer le port :

```
const express = require("express");
const app = express();
const Knex = require('knex')
const knexConfig = require('./knexfile')
const { Model, ForeignKeyViolationError, ValidationError } = require('objection')


// Initialize knex.
const knex = Knex(knexConfig.development)

// Bind all Models to a knex instance.
Model.knex(knex)

const backendPort = process.env.BACKEND_PORT;
```

Extrait du fichier server.js

Le serveur est prêt à être lancé, grâce à la commande *node server.js.*

## c. Développement de la base de données

Objection permet de créer des modèles qui correspondent à chacune de nos entités.
On peut y définir les relations,
les associations avec les autres Entités grâce à la propriété *relationMappings.*

La seule propriété requise dans un modèle est la propriété *tableName( ).*

```
const { Model } = require('objection');

class Course extends Model {
    static get tableName() {
        return 'course';
    }

    static get relationMappings() {
        const User = require('./User');
        const Step = require('./Step');
        const RatingCourse = require('./RatingCourse');
        const Tag = require('./Tag');

        return {
            user: {
                relation: Model.BelongsToOneRelation,
                modelClass: User,
                join: {
                    from: 'course.id_user',
                    to: 'user.id'
                }
            },
            steps: {
                relation: Model.HasManyRelation,
                modelClass: Step,
                join: {
                    from: 'course.id',
                    to: 'step.id_course'
                }
            },
            ratings: {
                relation: Model.HasManyRelation,
                modelClass: RatingCourse,
                join: {
                    from: 'course.id',
                    to: 'rating_course.id_course'
                }
            },
            usersbuy: {
                relation: Model.ManyToManyRelation,
                modelClass: User,
                join: {
                    from: 'course.id',
                    through: {
                        from: 'course_user_buy.id_course',
                        to: 'course_user_buy.id_user'
                    },
```

Extrait du modèle Course.js

La création et la mise à jour de la base de données se fait via la création de fichiers de migrations.

Grâce à la commande *npx knex migration:make [nom_de_la_migration]*, on crée un fichier qui contient le code qui permet d'upgrade et de downgrade la base de données. J'ai ainsi créé un fichier de migration par table.

```
exports.up = function(knex) {
    return knex.schema
        .createTable('course', table => {
            table.increments('id').primary();
            table.string('crs_name', 100);
            table.text('crs_description');
            table.string('crs_teaser', 300);
            table.float('crs_price');
            table.integer('id_user').unsigned().notNullable();
        })
};

exports.down = function(knex) {
    return knex.schema
        .dropTableIfExists('course');
};
```

Fichier de migration permettant la création de la table « course »

J'ai créé un dernier fichier qui vient modifier les tables afin d'ajouter les clés étrangères comme suit :

```
exports.up = function(knex) {
    return knex.schema
    .alterTable('user', table => {
        table.foreign('id_subscription').references('id').inTable('subscription').onDelete('RESTRICT');
    })
    .alterTable('course', table => {
        table.foreign('id_user').references('id').inTable('user').onDelete('RESTRICT');
    })
    .alterTable('capsule', table => {
        table.foreign('id_poi').references('id').inTable('poi').onDelete('RESTRICT');
```

Extrait du fichier de migration permettant la création des clés étrangères

Une fois tous les fichiers crées, j'ai mis à jour la base de données en appliquant tous les fichiers de migrations qui n'ont pas encore été appliqués, grâce à la commande *npx knex migrate:latest*. Afin de gagner du temps, j'ai exporté le fichier SQL de la base de données de l'application web, que j'ai modifié afin qu'il corresponde à la nouvelle architecture de ma base de données, que j'ai importé dans celle-ci, afin d'avoir des données fictives à exploiter lors du développement de l'API.

## d. Développement des routers et controllers

La base de données mise à jour, j'ai pu commencer à développer les controllers qui vont me permettre d'accéder aux données stockées dans la base de données. J'ai donc créé un dossier "controllers", dans lequel j'ai créé un fichier pour chaque entité qui sera concernée par les requêtes.
J'ai effectué la même opération en créant un dossier "routes" qui lui, contiendra un fichier pour chaque entité dans lequel seront précisées les routes, c'est-à-dire les différentes URL auxquelles seront renvoyées les réponses des requêtes contenues dans les fichiers controllers.

Par exemple, voici le controller courses.js qui contient toutes les requêtes concernant les parcours. La fonction getCourseTags( ) retourne tous les tags d'un parcours passé en paramètre. Cette fonction est asynchrone grâce au "async". La structure "try...catch" permet,

s'il y a une erreur lancée avec "await", de capturer celle-ci et de renvoyer un message d'erreur avec un statut 500.

```
const Course = require('../models/Course');
const Poi = require('../models/Poi');
const RatingCourse = require('../models/RatingCourse');
const Step = require('../models/Step');
const Tag = require('../models/Tag');
const Polyline = require('@mapbox/polyline');

const getCourses = async (req, res) => {
  try {
    const courses = await Course.query()
      .select('course.*', 'user.usr_name', 'poi.poi_name', 'poi.poi_address', 'poi.poi_latitude', 'poi.poi_longitude')
      .leftJoin('step','step.id_course', 'course.id')
      .leftJoin('poi', 'step.id_poi', 'poi.id')
      .leftJoin('user', 'course.id_user', 'user.id')
    return res.status(200).json(courses);
  } catch (e) {
    console.log(e);
    return res.status(500).send("getCourses() Error");
  }
};

const getCourseTags = async (req, res) => {
  try {
    const courseTags = await Tag.query()
      .select('tag.*')
      .innerJoin('course_tag', 'course_tag.id_tag', 'tag.id')
      .where('course_tag.id_course',req.params.id)
    return res.status(200).json(courseTags);
  } catch(e) {
    console.log(e);
    return res.status(500).send("getCoursesTags() Error");
  }
};
```

Extrait du controller de l'entité course

Ensuite, dans le router courses.js, j'ai créé une route, accessible avec la méthode "GET", qui accède au résultat de la requête effectuée par la fonction getCourseTags( ) :

```
const coursesController = require("../controllers/courses");

//Get course's tags
router.get("/:id/tags", coursesController.getCourseTags);
```

Extrait du router de l'entité course

Ainsi, en précisant dans le fichier server.js :

```
const courses = require("./routes/courses");
app.use("/courses", courses);
```

Extrait du fichier server.js

Le résultat de la requête est accessible à l'URL : [url_BDD]/courses/[id]/tags.

# 3. Développement de la partie front-end

## a. Mise en place de l’environnement de développement

J’ai donc choisi de développer à l’aide du framework Expo, qui apporte des outils qui facilitent le développement en React Native.
Expo étant déjà installé sur mon ordinateur, j'ai initialisé le projet grâce à la commande *expo init [nom_du_projet].* Les dossiers propres à Expo sont créés, le dossier node_modules qui comprend toutes les librairies nécessaires au fonctionnement du projet, le fichier App.js qui sera le point d'entrée de l'application, ainsi que les fichiers app.json, babel.config.js, package-lock.json et package.json qui sont différents fichiers de configuration.
Une fois le projet initialisé, j'ai pu commencer le développement.

## b. La navigation au sein de l’application

La navigation au sein d'une application mobile est différente d'une navigation sur un site web : les écrans d'une application mobile, appelés "screen" s'empilent les uns sur les autres, comme une pile d'assiette.

J'ai installé la librairie react-navigation qui s'occupe de gérer le routing et la navigation au sein d'une application mobile, ainsi que les dépendances associées :

- *npm install @react-navigation/native*
- *expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-safe-area-context @react-native-community/masked-view*

La fonction "createStackNavigator( )" retourne un objet qui contient 2 propriétés : Screen et Navigator, qui sont des composants React. Le Navigator contient des composants enfants Screen afin de définir la configuration des routes. Le composant NavigationContainer gère l'arborescence de la navigation et contient les variables d'état de la navigation.

Dans mon fichier App.jsx, j'ai donc créé cette arborescence :

```
<NavigationContainer>
  <Stack.Navigator
    screenOptions={{
      headerShown: false,
    }}
    >
    <Stack.Screen name='Tutoriel' component= {TutoScreen} />
    <Stack.Screen name="Home" component= {NavBar} />
    <Stack.Screen name='DisplayParcours' component= {DisplayParcours} />
    <Stack.Screen name='Sign In' component={SignInScreen} />
    <Stack.Screen name='Log In' component={LogInScreen} />
    <Stack.Screen name='User Profile' component={UserProfile} />
    <Stack.Screen name='Formules' component={FormulesScreen} />
    <Stack.Screen name='Choose Login' component={ChooseLogin} />
  </Stack.Navigator>
</NavigationContainer>
```

Arborescence de la navigation au sein de l'application

Ainsi, tous mes screens seront déclarés à l'intérieur du composant <Stack.Navigator>, grâce à un composant <Stack.Screen> qui possède 2 propriétés

- "name" qui correspond au nom de la route
- "component" qui correspond au composant à charger

J'ai ensuite procédé à la mise en place de la barre de navigation en installant la librairie « bottom-tabs » grâce à la commande *npm install @react-navigation/bottom-tabs.*
J'ai créé un composant <NavBar> dans lequel j'importe la fonction createBottomTabNavigator( ). Ensuite, j'ai procédé avec la même logique que la navigation : un composant <Tab.Navigator> qui contient un composant enfant <Tab.Screen> pour chaque bouton de la navbar. Les diverses propriétés de <Tab.Navigator> permettent de préciser la route par défaut, de modifier les icônes et la couleur de la navbar.
J'ai importé ensuite le composant <NavBar> dans le fichier App.jsx, puis l'ai défini comme un screen à part entière. Cela me permet de n'afficher la barre de navigation que lorsque je suis sur un des 4 écrans qui la composent.

```
const NavBar = () => {

    return (
        <Tab.Navigator
            initialRouteName={"Parcours"}
            screenOptions={({ route, navigation}) => ({
                tabBarIcon: ({focused}) => {
                    if (route.name === 'QR Code') {
                        return <MaterialCommunityIcons name="qrcode-scan" size={28} color="#fff" />;
                    }
                    else if (route.name === 'Parcours en cours') {
                        return <EnCours width={35} height={35} />;
                    }
                    else if (route.name === 'Parcours') {
                        return (
                            <View style={styles.skimtrackContainer}>
                                <SkimtrackLogo style={styles.skimtrack}/>
                            </View>
                        )
                    }
                    else if (route.name === 'Favoris') {
                        return <AntDesign name="heart" size={30} color="#fff" />
                    }},
            })}
            tabBarOptions={{
                activeTintColor: "#fff",
                inactiveTintColor: "#fff",
                showLabel: false,
                style: {
                    backgroundColor: "#FE6D64",
                    borderWidth: 1,
                    borderTopLeftRadius: 22,
                    borderTopRightRadius: 22,
                    borderColor: "#FE6D64",
                    padding: 0,
                },
                keyboardHidesTabBar: true
            }}
            >
            <Tab.Screen name='QR Code' component={ExploreScreen} />
            <Tab.Screen name='Favoris' component={ExploreScreen} />
            <Tab.Screen name='Parcours en cours' component={ExploreScreen} />
            <Tab.Screen name='Parcours' component={ExploreScreen} />

        </Tab.Navigator>
    );
```

Extrait du composant <NavBar> qui gère l'affichage de la barre de navigation

## c. Les fonts

Grâce à Expo, il est possible d'utiliser les polices de Google Font très facilement en installant la police choisie, par exemple "Montserrat" que j'utilise au sein du projet :
*expo install expo-font @expo-google-fonts/montserrat.*

Puis dans le fichier App.jsx, j'importe les polices dont j'ai besoin depuis le dossier *'@expo google-fonts/montserrat'.* Enfin, j'utilise le hook "useFonts( )" pour charger les polices au lancement de l'application. Désormais, j'ai accès à ces polices depuis n'importe quel fichier de mon projet.

```
let [fontsLoaded] = useFonts({
  Montserrat_200ExtraLight,
  Montserrat_300Light,
  Montserrat_400Regular,
  Montserrat_700Bold,
  Montserrat_500Medium,
  Montserrat_600SemiBold,
  RopaSans_400Regular,
  RobotoCondensed_400Regular
});

if (!fontsLoaded) {
  return <AppLoading />;
}
```

Extrait du fichier App.jsx

## d. Les contextes

J’ai utilisé les contextes qui me permettent de faire passer des données à travers l’arborescence sans avoir à les passer manuellement de composant en composant. Je peux donc accéder aux variables d’un contexte depuis n’importe où dans l’application. Ainsi lorsqu’une variable d’un contexte est modifiée, tous les composants qui en dépendent se mettent à jour.

**La géolocalisation :**
A son arrivée dans l'application, l'utilisateur est géolocalisé afin d'afficher sa position sur la carte et d'adapter l'affichage en fonction du lieu où il se trouve. Pour lui demander l'autorisation et récupérer sa position, j'ai mis un place un contexte React. Dans un nouveau fichier GeolocationContext.jsx, je crée le contexte GeolocalisationContext grâce à la méthode createContext( ). Ensuite, je développe le contenu du composant <Geolocation.Provider> de ce contexte :

- J'utilise le hook d'effet "useEffect( )" avec un tableau de dépendance vide, afin d'exécuter le code à l'intérieur uniquement lorsque le contexte est appelé la première fois. A l'intérieur du useState( ), je demande l'autorisation à l'utilisateur grâce à la méthode Location.requestPermissionsAsync( ), puis récupère la position de l'utilisateur grâce à la méthode Location.getCurrentPositionAsync(). A l'aide du hook useState( ), je viens stocker la latitude et la longitude dans la variable "location"
.
- Dans le return, je peux assigner au provider la valeur de la variable "location" grâce à la propriété "value". Ainsi, dans mon fichier App.jsx, je peux englober l'intégralité de mon <NavigationContainer> dans le composant <GeolocationProvider>. Je peux alors utiliser les variables contenues dans ce provider dans n'importe quel composant se trouvant en dessous de lui dans l'arborescence de la navigation, et ce, à l'aide du hook useContext : *const {location} = useContext(GeolocationContext);*

**L’utilisateur :**
J'ai également créé un contexte pour stocker l'utilisateur qui est connecté. Dans le composant <UserProvider>, j'utilise également un useEffect() qui appelle la fonction getUser. Cette fonction récupère les informations de l'utilisateur enregistré (donc s'il y a un token enregistré). La requête effectuée à l'API contient le token dans le header, car j'ai fait en sorte que cette route soit protégée par le middleware lors du développement de l'API.
Cette fois-ci, le tableau de dépendance contient la variable "token", afin que la fonction getUser() soit exécutée lorsque le contexte est chargé la première fois et à chaque fois que la variable "token" change.

Dans le return de mon provider, je peux assigner la valeur de la variable "values" grâce à la propriété "value". Ainsi, dans mon fichier App.jsx, je peux englober l'intégralité de mon <NavigationContainer> dans le composant <UserProvider>.
Je peux alors utiliser les variables user, token et les méthodes qui permettent de les modifier, dans n'importe quel composant dans l'arborescence, toujours à l'aide du hook useContext() : *const { user } = useContext(UserContext)*, et ainsi accéder par exemple au nom de l'utilisateur avec : "user.usr_name". L'affichage au sein de l'application est entièrement dynamique car il s'adapte aux données de chaque utilisateur.

```
import React, {useState, useEffect, createContext} from 'react';
import axios from 'axios';
import { backend } from '../conf';

export const UserContext = createContext(null);

export const UserProvider = (props) => {

    const [user, setUser] = useState({});
    const [token, setToken] = useState("");

    const values = {
        user, setUser, token, setToken
    };

    const getUser = async () => {
        if(token){
            const res = await axios.get(`${backend}/users/profile`,
            { headers : { Authorization : `Bearer ${token}`}});
            setUser(res.data);
        }
    }

    useEffect(() => {
        getUser();
    }, [token]);

    const {children} = props;

    return (
       <UserContext.Provider value={values} >
           {children}
       </UserContext.Provider>
    )
};
```

UserContext.js

## e. Écrans développés

Voici le détail que quelques écrans que j'ai pu développer jusqu'à présent. Les captures d'écran sont disponibles en annexe.

**OnBoarding**
Lorsque l'utilisateur arrive dans l'application, il arrive sur un écran d'accueil qui lui explique le principe de l'application, à l'aide d'un carrousel. Une fois à la fin du carrousel, lorsque l'utilisateur scrolle à nouveau, il arrive sur l'écran de login.
J'ai développé le carrousel en utilisant le composant React Native <ScrollView> qui permet d'afficher des éléments d'une liste sous la forme d'une liste scrollable.

J’ai rencontré une difficulté dans le développement de cet écran. En effet, lorsqu’on est sur le dernier élément du carrousel et que l’on scrolle à nouveau, on est dirigé vers le prochain écran.
Cependant, la propriété "onScroll" du composant <ScrollView> ne peut pas intervenir parce qu'il n'y a pas véritablement de scroll au niveau de la liste.
J'ai donc utilisé la libraire *react-native-gesture-handler* qui permet de capturer les gestes de l'utilisateur sur l'écran, et plus particulièrement le composant <PanGestureHandler>. J'ai également utilisé une fonction trouvée sur le forum « StackOverFlow », qui me permet de savoir si l'élément affiché à l'écran est le dernier de la liste.

Je capture donc l'événement grâce à la propriété "onHandlerStateChange" qui appelle la fonction handleEvent( ). Celle-ci récupére l'évenement, et vérifie que la translationX soit inférieure à -200, ce qui correspond à un scroll horizontal de la gauche vers la droite. Si en plus, la variable {endView} est vraie, ce qui signifie qu'on est sur le dernier élément de la liste, alors on navigue vers l'écran dont la route possède la propriété name "Choose Login".
Pour cela, j'ai utilisé la méthode navigation.reset afin de réinitialiser l'historique de navigation, et d'empêcher un utilisateur sous Android de revenir à l'écran précédent avec le bouton retour arrière. L'écran "Choose Login" devient le premier écran de l'application.

## Login

L'écran "Choose Login" permet à l'utilisateur de choisir de se connecter, de s'inscrire ou continuer sans être authentifié.

### Inscription

J'ai utilisé les librairies Formik et Yup pour la création du formulaire et la validation des données saisies par l'utilisateur. J'ai développé mon formulaire d'inscription au sein d'un composant <Formik>. Ce composant comporte plusieurs propriétés :

- initialValues, qui est un objet qui contient le nom de chaque champ ainsi que leur valeur initiale.
- onSubmit, qui contient le code à exécuter lorsque l'utilisateur soumet le formulaire.
- validationSchema, qui contient un objet yup, qui indique les différents champs, leur type, s'ils sont requis, leur valeur minimale etc., ainsi que les messages d'erreur à afficher pour chaque champ. Je peux ainsi procéder à une première validation des données côté client.

Lorsque l'utilisateur soumet le formulaire et qu'il n'y a pas d'erreur côté client, j'effectue une requête POST auprès de l'API pour enregistrer les données, grâce à axios.
Si elle renvoie un message d'erreur, par exemple parce que l'adresse email existe déjà en base de données, je récupère ce message, le stocke dans la variable "errorMessage" et l'affichage en dessous de mon formulaire. L'utilisateur est alors informé de l'échec de son inscription et sait quel champ il doit modifier.

Connexion

Le formulaire de connexion fonctionne de la même manière que le formulaire d'inscription. J'effectue une première vérification des données côté client grâce au schéma et au yup.object qui contient mes vérifications.
Lorsque l'utilisateur soumet le formulaire et qu'il n'y a pas d'erreur côté client, j'effectue une requête de type POST auprès de l'API pour vérifier l'identité de l'utilisateur. Si les informations ne sont pas valides, je récupère le message d'erreur renvoyé par l'API afin de l'afficher à l'utilisateur. Si l'authentification réussit, je récupère le token qu'elle me renvoie, le stocke et dirige l'utilisateur au sein de l'application.

**Liste des parcours + carte**

Une fois connecté, l'utilisateur arrive sur un écran qui contient une carte, une barre de recherche ainsi qu'une liste scrollable de parcours.
J'ai divisé cet écran en plusieurs composants, créés dans des fichiers séparés afin de gagner en visibilité et facilité la maintenance du code :

- <HeaderMap> qui affiche la barre de recherche et l'avatar de l'utilisateur
- <Map> qui gère l'affichage de la carte, grâce à la librairie *react-native-maps*, basée sur les services de Google Maps où il est possible d'afficher la position de l'utilisateur. Grâce à une requête à l'API, je récupère les points d'intérêts du parcours qui apparait à l'écran, que j'affiche grâce au composant <Marker> intégré à la librairie.
  Pour dessiner la route entre ces points, j'ai effectué une requête à l'API Google Directions. Voilà à quoi ressemble une requête :
  *https://maps.googleapis.com/maps/api/directions/json?origin=[coord_origin]&destination=[coord_dest]&waypoints=[coord_point1]|[coord_point2]|[coord_point3]&mode=walking&key=[API_KEY]*
  Parmi les nombreuses données renvoyées par l'API Google Directions, il y a une chaîne de caractères qui, une fois décodée, correspond à la liste des coordonnées (désignées par une latitude et une longitude), qu'il faut

emprunter pour passer par tous les points d'intérêts. Je stocke donc cette ligne encodée en base de données, et je la décode en back grâce à la librairie *@mapbox/polylines* avant de retourner le résultat de la requête. Une fois la liste obtenue, j'utilise le composant <Polyline> qui prend en paramètre mon tableau de coordonnées, et qui dessine la route entre chaque point d'intérêt.

- <ButtonsMap> qui affiche le bouton filtre et le bouton pour lancer un parcours guidé
- <ParcoursList> qui gère l'affichage d'une liste scrollable grâce au composant <FlatList>. Pour cela, je récupère tous les parcours en base de données que je stocke dans un tableau, puis je parcours le tableau à l'aide de la méthode map( ). Ainsi, pour chaque item, je retourne un <Parcours> qui est un composant externe, à qui je transmets l'id du parcours qu'il rend.

De plus, j'avais besoin de savoir quel parcours de la liste apparait à l'écran lorsque l'utilisateur scrolle, afin de modifier la couleur de sa bordure, et d'envoyer cette information au composant parent, afin qu'il l'envoie au composant <Map> qui va se charger de l'appel à l'API afin d'afficher les points d'intérêts et de dessiner la route entre eux. Après plusieurs recherches sur les forums dédiés et sur la documentation de React Native, j'ai utilisé 2 propriétés du composant <FlatList> : viewabilityConfig et onViewableItemsChanged. La première me sert à spécifier quand un item est considéré comme affiché à l'écran.

```
// //know which course is displayed
const viewConfigRef = useRef({ viewAreaCoveragePercentThreshold: 80 , minimumViewTime: 400})
const onViewRef = useRef((viewableItems)=> {
    setCourseDisplay(viewableItems.changed[0].key);
})
```

Extrait du composant <ParcoursList>

Ici, il doit couvrir 80% de la largeur de l'écran et resté à l'écran au moins 400ms. Lorsque c'est le cas, j'appelle la fonction définie au sein de la propriété onViewableItemsChanged, qui assigne l'id de la course affichée à la variable « courseDisplay ».

### Profil utilisateur

La page Profil de l'utilisateur récupère le contexte UserContext afin de récupérer les informations de l'utilisateur, y compris son token de session

```
const { user } = useContext(UserContext);
const { token } = useContext(UserContext);
```

Extrait du composant <UserProfileScreen>

Ce dernier est utilisé lors de la requête POST qui permet la déconnexion de l'utilisateur. Dans ce cas, il est renvoyé à la page de connexion et la navigation est réinitialisée grâce au navigation.reset( ).

# Fonctionnalité la plus représentative: L'authentification

## 1. Jeu d'essai

L'utilisateur a besoin d'être connecté pour avoir pleinement accès à certaines fonctionnalités. Il fallait donc constituer une passerelle entre l'API et le front pour authentifier l'utilisateur, avec une attention toute particulière sur la sécurisation du processus.

Les jetons JWT, de par leurs signatures construites grâce à une clé privée détenue par le serveur, semble être une solution tout indiquée.
Plus de détails à ce sujet seront présents dans ma veille anglophone, disponible en page 28.

Tout d'abord, j'ai installé la librairie "jsonwebtoken" avec la commande *npm install jsonwebtoken.* Elle permet de générer et de décoder des tokens lors du processus d'authentification dans le cadre du protocole OAuth2.

Dans le modèle User, j'ai créé une fonction asynchrone qui permet d'attribuer un token à un utilisateur grâce à la méthode jwt.sign( ) qui prend en paramètres l'id de l'utilisateur, la clé secrète qui sera nécessaire au décodage, et le délai d'expiration du token. Ensuite, j'effectue une requête à la base de données pour ajouter le token dans la table "user".

```
//Generate token
generateAuthToken = async () => {
    const user = this;
    const token = jwt.sign({ id : user.id.toString()}, 'jeTesteLesTokens', { expiresIn: '7 days' });
    await User.query().patchAndFetchById(user.id, {usr_token: token});
    return token
}
```

Extrait du modèle user

Pour des raisons de sécurité, il est nécessaire que le mot de passe des utilisateurs ne soit pas stocké en clair dans la base de données. J'ai donc installé et utilisé la librairie "objection-password" qui permet automatiquement de générer un hash du mot de passe avant de le stocker lorsqu'il concerne un champ d'un modèle Objection.

Ensuite, j'ai créé le CRUD pour l'entité User :

Create : création d'un nouvel utilisateur avec la méthode POST.
Lorsqu'un utilisateur s'inscrit, la fonction récupère les données envoyées qui sont stockées dans le *req.body*.

Elle effectue une requête en base de données afin de vérifier si l'email ou le nom d'utilisateur existent déjà. Si c'est le cas, elle renvoie un message que je peux récupérer côté front. Sinon, elle insère les données dans la base de données (dont le hash du mot de passe), génère un token grâce à la fonction generateAuthToken( ), qui est donc stocké, et retourne ce nouvel utilisateur. Afin que la réponse envoyée ne contienne ni le mot de passe ni le token, j'ai ajouté la fonction suivante au modèle User :

```
//Get Public Profile without password and token
toJSON = function () {
    const user = this;
    delete user.usr_password;
    delete user.usr_token;
    return user;
}
```

Extrait du modèle user

Read : récupération les informations d'un utilisateur lors de sa connexion avec la méthode GET.
Lorsque l'utilisateur se connecte, je fais une requête en base de données afin de savoir si l'adresse email contenue dans le *req.body.usr_email* existe.
Si ce n'est pas le cas, je renvoie un message qui annonce l'échec de l'authentification, en demandant à l'utilisateur de vérifier ses informations. Sinon, je décode le mot de passe contenu dans le *req.body* grâce à la méthode verifyPassword( ).
S'il ne correspond pas, j'affiche le même message que précédemment, sans donner d'informations sur le champ invalide, pour des raisons de sécurité.
Si l'authentification réussit, je génère un token puis effectue une nouvelle requête afin de récupérer les nouvelles données de l'utilisateur, que je renvoie, ainsi que le token.

Update : mise à jour des informations d'un utilisateur avec la méthode PATCH.
Les données que l'utilisateur souhaite modifier sont contenues dans le *req.body* sous la forme d'un objet contenant un ensemble de couples clé-valeur. Je récupère les clés grâce à la méthode Object.keys( ) puis les compare avec les noms des champs qu'il est possible de modifier. Si l'utilisateur est autorisé à effectuer ces modifications, je vérifie qu'il existe bien en base de données, puis applique les changements et retourne les informations de l'utilisateur.

Delete : suppression d'un utilisateur avec la méthode DELETE.
J'effectue une requête à la base de données pour récupérer l'utilisateur et tenter de le supprimer. La méthode deleteById( ) retourne le nombre de lignes supprimées. Si "user" vaut 0, "!user" vaut 1, je renvoie donc une erreur 404. Sinon, c'est que la suppression a fonctionné, je renvoie donc le message "user deleted".

Ainsi que la fonction qui permet la déconnexion avec la méthode POST :

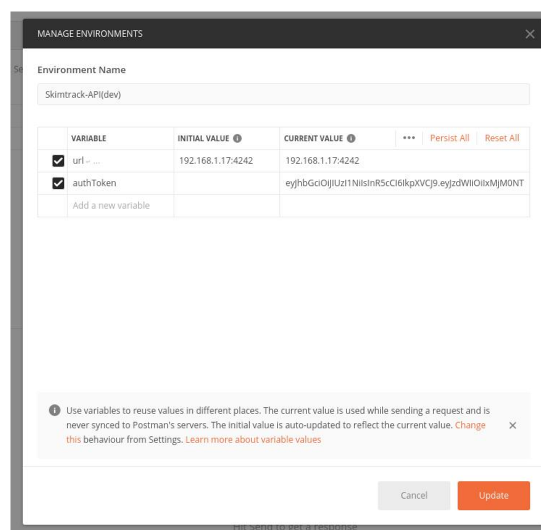
Je vérifie que le token envoyé avec la requête correspond à celui stocké en base de données. Si c'est le cas, j'efface le token stocké, puis renvoie le message "User disconnected".

Une fois cela fait, j'ai créé un nouveau dossier "middleware", dans lequel j'ai créé le fichier « auth.js » qui contient la fonction auth( ) qui permet de vérifier l'authentification de l'utilisateur lors de chaque appel à l'API.
En effet, lors de chaque appel, le token est envoyé dans le header de la requête. Une fois récupéré, il est décodé grâce à la méthode jwt.verify( ) qui prend en paramètre le token ainsi que la clé utilisée pour l'encoder. Je récupère ainsi un id d'utilisateur, et j'effectue une requête à la base de données afin de vérifier qu'un tel utilisateur existe et que le token correspond bien à celui envoyé. Si c'est le cas, j'autorise l'appel à l'API grâce à la méthode next( ), sinon un message d'erreur est envoyé : "Please authenticate".
Je peux désormais restreindre l'accès à certaines routes de l'API en ajoutant le middleware "auth" : par exemple :

```
//Get user profile
router.get('/profile', auth, usersController.getUserProfile);
```

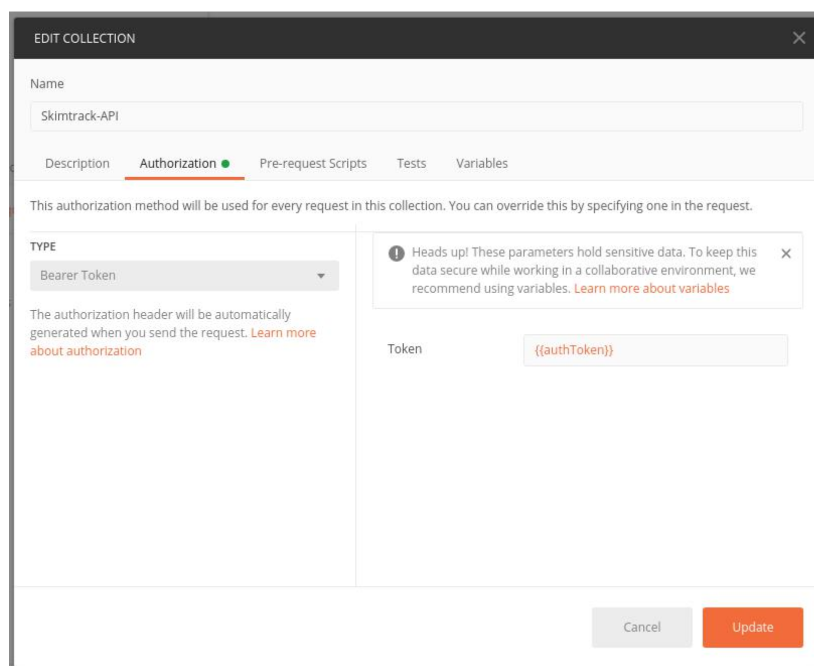
Extrait du router user

Pour accéder à la route "users/profile", il faudra que l'utilisateur soit authentifié.
Grâce au logiciel Postman, il est possible de tester les différentes routes et d'implémenter la gestion des tokens, notamment grâce aux variables d'environnement.
Dans un premier temps, j'ai créé les variables "url" et "authToken". La variable "url" me permet de stocker l'url de l'API, afin de ne pas avoir à changer l'adresse dans toutes les url si celle-ci change. Je peux désormais utiliser la syntaxe "{{url}}/users" par exemple. Quant à la variable "authToken", elle me permettra de stocker le token de l'utilisateur et de l'ajouter au header de mes requêtes au sein de Postman.




Variables d'environnement Postman

J'ai créé ensuite une collection, appelée « Skimtrack-API », en précisant dans l'onglet "Authorization" qu'il s'agit d'un type "Bearer Token" et que le Token est égal à la variable {{authToken}}.




Paramètres de la collection Skimtrack-API

J'ai créé ensuite la plupart de mes requêtes API, notamment celles qui concernent l'utilisateur. Je sais qu'un utilisateur reçoit un token lorsqu'il s'inscrit ou se connecte (requêtes "Create User" et "Login User" dans Postman). Dans l'onglet "Tests" de ces requêtes, j'ai ajouté du code JavaScript qui me permet d'update la variable "authToken" avec le token renvoyé par l'API si la réponse possède un statut 200). Ainsi, il est possible de tester l'intégralité des URL de l'API, afin de détecter une possible erreur dans la gestion de l'authentification des utilisateurs.

POST {{url}}/users/login

Params Authorization Headers (8) Body Pre-request Script Tests Settings

```
1  if (pm.response.code === 200) {
2      pm.environment.set('authToken', pm.response.json().token)
3  }
4
5
```

Code JavaScript dans l'onglet « tests » de la requête « Login User »

## 2. Recherches anglophones sur le JWT

Ayant très peu abordé la notion d'authentification et de sécurité d'une API lors de la formation, je me suis documentée sur le principe du JSON Web Token (ou JWT). Voici un extrait du site officiel jwt.io, ainsi que la traduction.

"In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header : The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- Payload : The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims. Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.
- Signature : To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is."

« Dans sa forme compacte, le JSON Web Token est composé de 3 parties séparées par un point, qui sont :

- le header : Constitué de 2 parties ; le type du token, qui est JWT, et l'algorithme utilisé pour l'encoder, tel que HMAC SHA256 ou RSA.
- le payload : Il contient des informations à propos d'une entité (qui est généralement un utilisateur), et des données supplémentaires. Il y a 3 types d'informations : les informations enregistrées, les informations publiques et les informations privées. Notez que pour les tokens signés, ces informations sont accessibles par n'importe qui, même si elles sont protégées contre la falsification. Ne stockez jamais d'informations secrètes dans le payload ou le header, à moins qu'elles soient encryptées.
- La signature : Pour créer la signature, il faut prendre le header et le payload encodés, un secret, l'algorithme spécifié dans le header et les signer. La signature est utilisée pour vérifier que le message n'a pas été changé entre temps, et dans le cas des tokens signés avec une clé privée, elle permet de vérifier que l'expéditeur du JWT est bien celui qu'il prétend être. »

# Conclusion

Concernant le projet, les fonctionnalités de base sont développées. Je suis à la moitié de mon stage, il me reste encore 3 mois pour continuer le développement afin de proposer une première version dont le déploiement est prévu au printemps 2021.

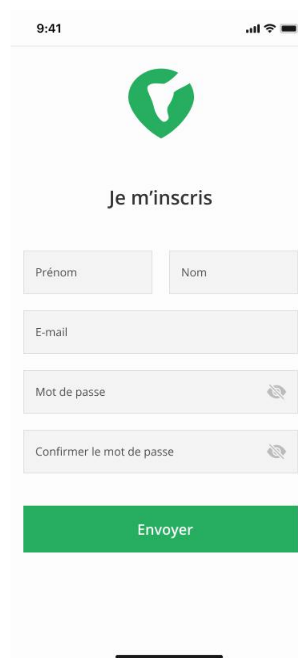
En acceptant ce stage, je savais qu'il serait un défi pour moi. Nous avions abordé les technologies comme React Native et Node.js lors de la formation, mais mes connaissances ne me permettaient pas à elles-seules de procéder au développement d'une application mobile assez facilement.

En effet, le développement est un défi de tous les jours, car il me faut apprendre à l'aide de la documentation officielle, à l'aide des forums dédiés, de tutoriels vidéo, et reproduire ce que je viens d'apprendre sur mon projet. Je pense que c'est la meilleure méthode de travail afin de progresser rapidement, et que c'est une bonne façon d'entrevoir ce qu'est le métier de développeur.

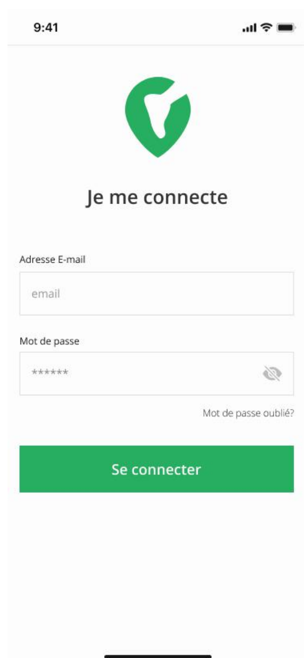
# Annexes

Annexe 1 – Écran d'onBoarding

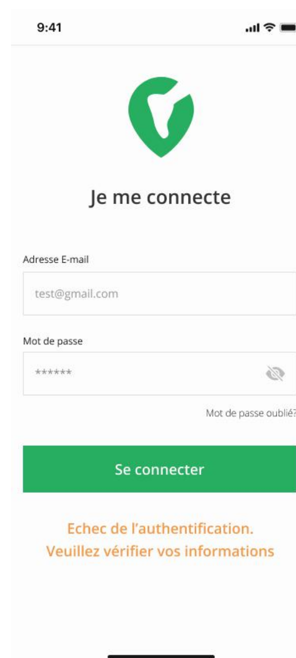
Annexe 2 – Écran d'inscription



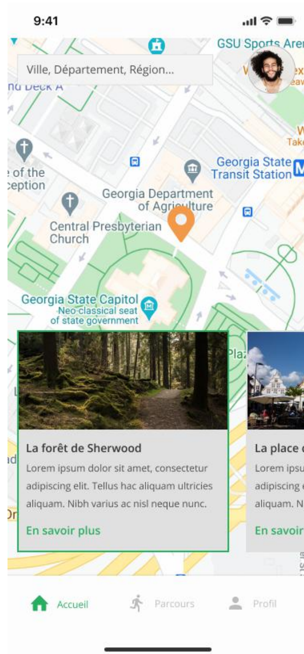

Annexe 3 – Écran de connexion



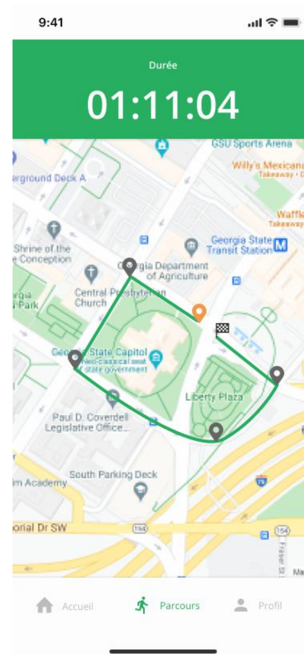

Annexe 4 – Écran de connexion erronée



Annexe 5 – Écran des parcours




Annexe 6 – Écran d'un parcours




Annexe 7 – Écran Profil utilisateur